Fundamentos da Programação
LEIC/LETI

Recursão de cauda

Aula 22

Alberto Abad, Tagus Park, IST, 2021-22

Recursão e Iteração

- No desenvolvimento de programas é importante ter em conta como é que um programa é executado e, em particular, como é que o processo computacional inerente à execução do programa evolui.
- Hoje, vamos a analisar alguns padrões típicos de evolução de programas e funções, em particular:
  - Recursão linear **(aula de ontem)**
  - Iteração linear **(apresentado em semanas anteriores)**
  - Recursão de cauda **(novo hoje)**
  - Recursão em árvore **(novo amanhã)**

Recursão Linear

- A recursão linear é a forma mais comum de recursão.
- Vários ambientes locais são gerados por causa da chamada repetida da pópria função: **expansão** de memória.
- Em cada ambiente ficamos com uma operação adiada até atingir o caso terminal, em que os ambientes vão sendo libertados e ocorre uma **contração**.

```python
def fatorial(n):
    if n == 0:
        return 1
    else:
        return n * fatorial(n - 1)
```

In [ ]:

```

```

Recursão Linear

Padrão de execução

```
fatorial(4)
| fatorial(3)
| | fatorial(2)
| | | fatorial(1)
| | | | fatorial(0)
| | | | return 1
| | | return 1
| | return 2
| return 6
return 24
```

- O número de ambientes cresce **linearmente** em função de um determinado valor da entrada → *Processo Recursivo Linear*
- Considerações sobre eficiência:
  - Tempo: linear, *O(n)*
  - Espaço: linear, *O(n)*

Iteração Linear

- Na iteração linear, gera-se um processo iterativo caracterizado por:
  - Um conjunto de variáveis de estado
  - Regras que especificam como actualizá-las.

```
def fatorial(n):
    res = 1
    for i in range(1, n+1):
        res = res * i
    return res
```

- O número de operações sobre as variáveis de estado cresce linearmente com um valor associado à função → *Processo Iterativo Linear*

- Considerações sobre eficiência:
  - Tempo: linear, *O(n)*
  - Espaço: constante, *O(1)*

```
In [ ]:
```

Recursão de Cauda

- É possível definir processos recursivos que utilizem variáveis de estado de forma semelhante aos processos iterativos → *Recursão de cauda*

- Na recursão de cauda:
  - Primeiro o cálculo é realizado e só depois é feita a chamada recursiva.
  - Na chamada são passados os resultados da etapa atual para a próxima etapa recursiva.
  - A chamada recursiva é a última operação realizada pela função, não existindo operações adiadas.

```
def fatorial_aux(n, res):
    if n == 0:
        return res
    else:
        return fatorial_aux(n - 1, n * res)
```

Recursão de Cauda

- Podemos organizar um pouco melhor a função como vimos na aula anterior:
  - Definimos uma função auxiliar.
  - O acumulador na função auxiliar é inicializado com o valor a retornar no caso base da recursão linear.

```
def factorial(n):
    def factorial_aux(n, acc):
        if n == 0:
            return acc
        else:
            return factorial_aux(n - 1, n * acc)
    return factorial_aux(n, 1)
```

In [2]:
```
def fatorial(n):
    # função auxiliar
    def fatorial_aux(n, acc):
        # Caso terminal
        if n == 0:
            return acc  # devolve resultado final
        else: # caso geral
            return fatorial_aux(n - 1, n * acc) # chamada função re
cursiva, atualização feita na passagem de parâmetros

    # chamada a função auxiliar, com valor inicial do argumento res
ultado igual ao valor inicial no processo iterativo
    return fatorial_aux(n, 1)

fatorial(5)
```

Out[2]: 120

Recursão de Cauda

Padrão de execução

```
fatorial(4)
| fatorial_aux(4, 1)
| | fatorial_aux(3, 4)
| | | fatorial_aux(2, 12)
| | | | fatorial_aux(1, 24)
| | | | | fatorial_aux(0, 24)
| | | | | return 24
| | | | return 24
| | | return 24
| | return 24
| return 24
return 24
```

- Como a recursão linear, o número de ambientes cresce **linearmente** em função de um determinado valor da entrada.
- Considerações sobre eficiência:
  - Tempo: linear, *O(n)*
  - Espaço: linear, *O(n)*

Recursão de Cauda e Iteração Linear

Vantagens da recursão de cauda

- A recursão de cauda pode facilmente ser convertida/optimizada numa iteração linear:

```python
In [7]: def fatorial(n, fac):
            while True:
                if n == 0:
                    return fac
                else:
                    return fatorial(n-1, n*fac) ## <---CHANGE HERE
                    # n, fac = n-1, n*fac


        fatorial(20, 1)
```

```
Out[7]: 2432902008176640000
```

- Algumas linguagens fazem a optimização da recursão de cauda para um processo iterativo automaticamente.
- O Python **não optimiza** as recursões de cauda.

Recursão e Iteração

Exercício 1, *potencia*

Recursão e Iteração

Exercício 1, *potencia*

In [13]:
```
def potencia_il(x, n):
    res = 1
    while n > 0:
        res = res * x
        n = n - 1
    return res


def potencia_rl(x, n):
    if n == 0:
        return 1
    else:
        return x * potencia_rl(x, n-1)


def potencia_rc(x, n):
    def potencia_aux(x, n, res):
        if n == 0:
            return res
        else:
            return potencia_aux(x, n-1, res*x)

    return potencia_aux(x, n, 1)


print(potencia_il(2,6))
print(potencia_rl(2,6))
print(potencia_rc(2,6))
```

```
64
64
64
```

Recursão e Iteração

Exercício 2, soma elementos duma lista

Recursão e Iteração

Exercício 2, soma elementos duma lista

In [9]:
```python
def soma_il(lst):
    soma = 0
    for i in range(len(lst)):
        soma = soma + lst[i]
    return soma

def soma_il(lst):
    final = 0
    for i in lst:
        final += i
    return final

def soma_rl(lista):
    # return 0 if not lista else lista[0] + soma_rl(lista[1:])
    if not lista:
        return 0
    else:
        return lista[0] + soma_rl(lista[1:])

def soma_rc(lst):
    def soma_aux(lst, res):
        # return res if len(lst) == 0 else soma_aux(lst[1:], res +
lst[0])
        if len(lst) == 0:
             return res
        else:
            return soma_aux(lst[1:], res + lst[0])

    return soma_aux(lst, 0)


print(soma_il([2,4, -1]))
print(soma_rl([2,4, -1]))
print(soma_rc([2,4, -1]))
```

```
5
5
5
```

Recursão e Iteração

Exercício 3, Exame 1 2018/19

6. Escreva a função `soma_n_vezes` que recebe três argumentos, `a`, `b` e `n`, e que devolve o valor de somar `n` vezes `a` a `b`, isto é, b + a + a + ... + a, `n` vezes. Não é necessário verificar a correção dos argumentos. A sua função não pode usar a operação `*`.

In [14]:
```python
def soma_n_vezes_il(a, b, n):
    for i in range(n):
        b += a
    return b

def soma_n_vezes_rl(a, b, n):
    if n == 0:
        return b
    else:
        return a + soma_n_vezes_rl(a, b, n-1)

def soma_n_vezes_rc(a, b, n):
    def soma_aux(n, res):
        if n == 0:
            return res
        else:
            return soma_aux(n-1, res+a)
    return soma_aux(n, b)

print(soma_n_vezes_il(2, 7, 3))
print(soma_n_vezes_rl(2, 7, 3))
print(soma_n_vezes_rc(2, 7, 3))
```

```
13
13
13
```

Recursão e Iteração

Exercício 4, Exame 2 2018/19

Considere a função, definida para inteiros não negativos, do seguinte modo:

$$f(n) = \begin{cases} 1 & \text{se } n = 0 \\ 2 \cdot f(n-1) & \text{se } n \text{ é par} \\ 3 \cdot f(n-1) & \text{se } n \text{ é impar} \end{cases}$$

Recursão e Iteração

Exercício 4, Exame 2 2018/19

Considere a função, definida para inteiros não negativos, do seguinte modo:

$$f(n) = \begin{cases} 1 & \text{se } n = 0 \\ 2 \cdot f(n-1) & \text{se } n \text{ é par} \\ 3 \cdot f(n-1) & \text{se } n \text{ é impar} \end{cases}$$

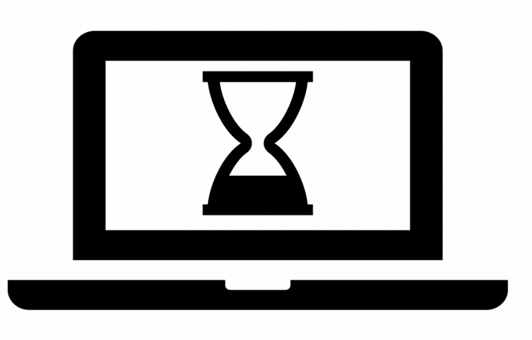
In [16]:

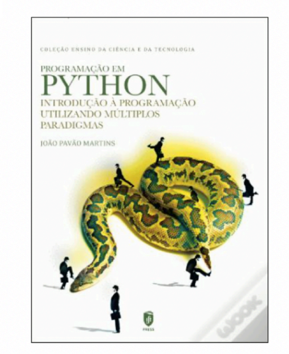
```python
# iterativa
def f_il(n):
    res = 1
    for i in range(1, n+1):
        if i % 2 == 0:
            res = 2 * res
        else:
            res = 3 * res

    return res


# operações adiadas
def f_rl(n):
    if n == 0:
        return 1
    else:
        # return f_rl(n-1) * (3 if n % 2 else 2)
        if n % 2 == 0:
            return 2 * f_rl(n-1)
        else:
            return 3 * f_rl(n-1)


# recursão da cauda
def f_rc(n):
    def f_aux(n, res):
        if n == 0:
            return res
        if n % 2 == 0:
            return f_aux(n-1, 2*res)
        return f_aux(n-1, 3*res)

    return f_aux(n, 1)

print(f_il(11))
print(f_rl(11))
print(f_rc(11))
```

```
23328
23328
23328
```

Tarefas próximas aulas

- Estudar matéria e completar exemplos
- A Ficha 5 da próxima semana é sobre **recursão**





In [ ]:

```
```